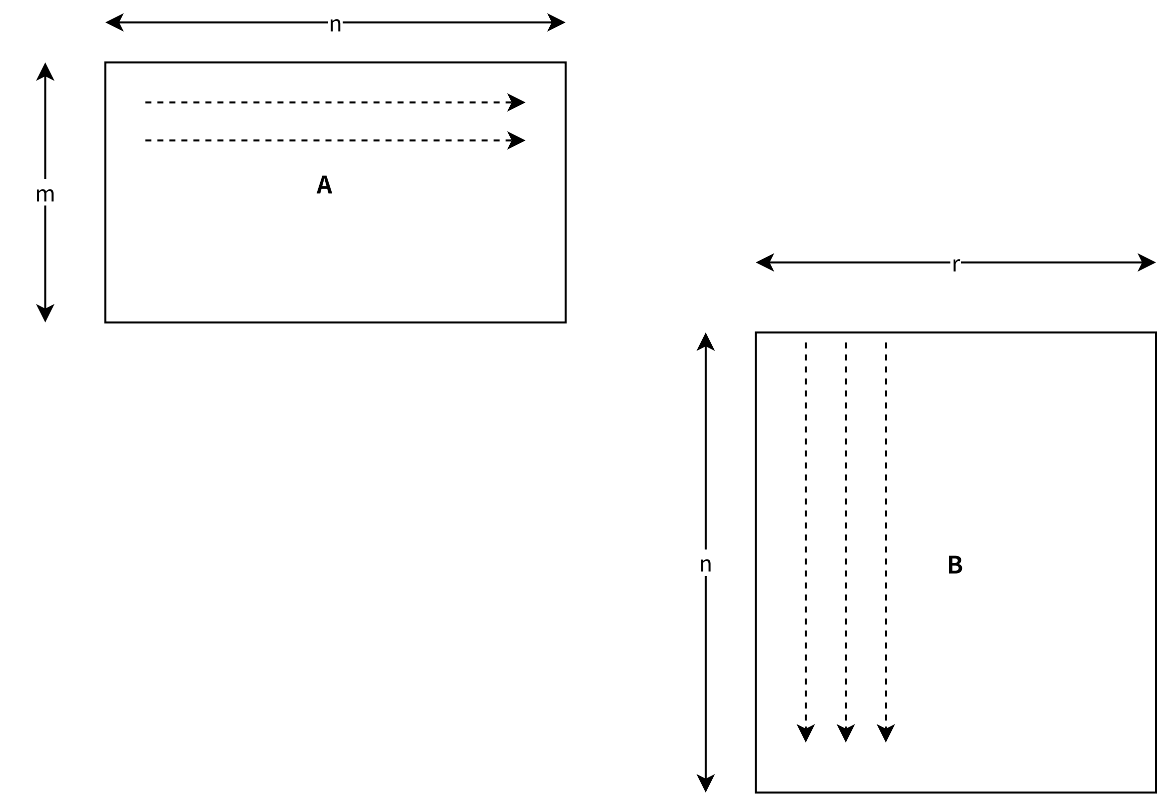
PESS第八课作业讲解

1. Assume we want to multiply two rectangular matrices: m × n with n × r. Given the same tall cache assumption, please analyze the complexity for one of the following four cases: the two cases for the naive approach (n > M/B and M/r < n < M/B), the block approach, and the cache-oblivious approach. **You may pick whichever case you want to analyze.**

```
void Mult(double *C, double *A, double *B, int64_t m, int n, int r) {
  for (int64_t i=0; i < m; i++)
    for (int64_t j=0; j < r; j++)
      for (int64_t k=0; k < n; k++)
        C[i*r+j] += A[i*n+k] * B[k*r+j];
}
```



> 1. $n > M/B$

> *n > M/B，意思是无法将每行的一个block都分配一个cache line，B矩阵每列n次miss。到尾部后从头遍历会继续触发新的miss，共r列。m行每次重复， 故综合cache miss为* $\Theta(n * r * m)$
>
> 2. *M/r < n < M/B*
>
>   *B矩阵大小n*r > M，说明缓存无法装进所有B矩阵元素。*
>
>   *n < M/B，说明缓存可以将每一行的一个块分配一个独立cache line*
>
>   *B矩阵一列n次miss，后续直到第B列均会cache hit，按照宽度r，共需r / B次触发cache miss。由于M/r < n，说明m行每次换行，需要重新遍历B矩阵，触发cache miss*
>
>   *综合cache miss为* $\Theta(n * r / B * m)$

2. **Tableau Construction.** In this problem, we are only interested in computing the final value of the tableau, stored in A(N-1,N-1), and hence we really only need 2N − 1 amount of space during computation. Thus, the algorithm declares A as an array of size 2N − 1.

   Explain why 2N − 1 space is sufficient and how the tableau function utilizes the 2N − 1 space.

```c
 #define A(i, j) A[N + (i) - (j) - 1]

void tableau(double *A, size_t N) {
  for (size_t i = 1; i < N; i++) {
    for (size_t j = 1; j < N; j++) {
        A(i, j) = f(A(i-1, j-1), A(i, j-1), A(i-1, j));
    }
  }
}


for (size_t i = 0; i < N; i++) {
    A(i, 0) = INIT_VAL;
}

for (size_t j = 0; j < N; j++) {
    A(0, j) = INIT_VAL;
}

tableau(A, N);
res = A(N - 1, N - 1);
```

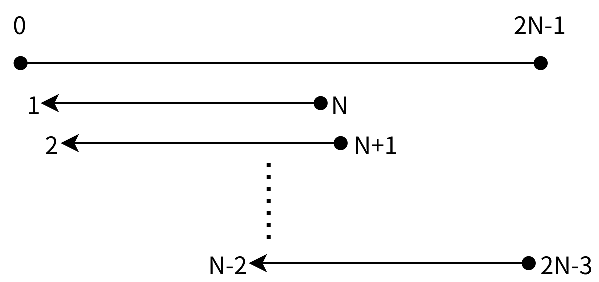
```

a.
#define A(i, j) A[N + (i) - (j) - 1]

由于只关心A[N-1, N-1]结果

所以i范围[0, N-1], j范围[0, N-1]
所以i - j范围[-(N-1), N-1]
所以 N + i -j - 1  = N - 1 + (i -j ) <= N - 1 + N - 1 = 2N - 2
所以最大下标2N - 2, 下标从0开始, 实际空间需要2N - 2 + 1 = 2N - 1

b.
两层循环, 先i, 再j,
(i,j)规律
[1,1] [1,2] [1, 3] ... [1, N-2]
[2,1] [2,2] [2, 3] ... [2, N-2]
[3,1] [3,2] [3, 3] ... [3, N-2]
...
[N-2,1] [N-2,2] [N-2, 3] ... [N-2, N-2]

i-j规律
0, -1, -2, ... -N + 3
1, 0, -1, ...  -N + 4
2, 1, 0, ...   -N + 5
...
N-3, N-4, N-5, ... 0

A(N-1,N-1)规律
N-1, N-2, N-3,... 2
N, N-1, N-2,...   3
N+1, N, N-1,... 4
...
2N-4,2N-5,2N-6,.. N-1

对于第一次访问, 先访问N-1, 再访问N, 再访问N-2, 再访问N-1

实际访问的内存范围如下:
N   ~  1
N+1 ~  2
N+2 ~  3
...
N+B ~  B+1
...
2N-3 ~ N-2
```

3. Recall the tall cache assumption, which states that $B^2 < \alpha M$, where B is the size of the cache line, M is the size of the cache, and $\alpha \leqslant 1$ is a constant.Assuming that an optimal replacement strategy holds and that the cache is tall,give a tight upper bound on the cache complexity Q(n) for each of the following cases using O notation, where $c \leqslant 1$ is a sufficiently small constant:

1. $n \geqslant cM$

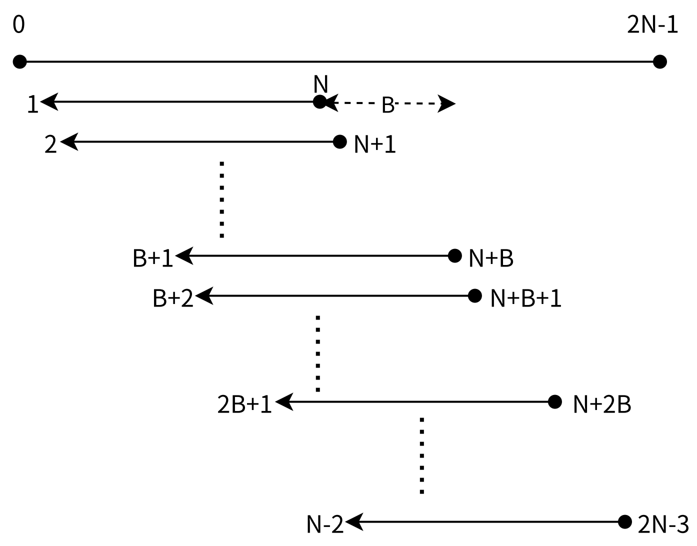
2. $n < cM$

> **n为table边长，实际需要内存空间2n。**
> 遍历内存按照倒序遍历，每次访问四次，三个地址，

> 1. *n >=cM，每次换行遍历时无法将整行放入cache，总cache miss：* $O(n^2/B)$
>
> 
>
>
> ***第一行***
> 访问1~n内存地址，cache line大小B。每B次触发一次cache miss。
> *总共miss:* $O(n/B)$
> ***第二行***
>
> *访问2~n+1，由于n > cM，说明第一行访问1地址时， N映射的cache line肯定被驱逐，因为最长时间没访问。所以新访问N+1必然触发cache miss，往后继续访问每隔B个会驱逐旧的cache line映射，触发cache miss。*
>
> ...
>
> *总共miss:*$O(n/B)$
>
> *所有行：* $O(n/B) \ * \ n \ = \ O(n^2/B)$

2. *n < cM，缓存可以装入整行且有余量，总cache miss：* $O(n/B)$

第一行 $O(n/B)$



第二行

访问2~n+1内存地址，在缓存中

总共miss: $O(0)$

第B行

极限情况下，地址N的cache line映射正好从N开始，则N~N+B-1已经放入缓存

第一次：N+B则miss(最优策略，[N+B]新映射cache line后，最远被使用的[N-B,N]范围使用的cache line会被驱逐)

后续均存在之前的缓存中

总共cache miss $O(1)$

第B+1行

同第二行

..

第2B行

同第B行

...

所有行cache miss： $O(n/B) + O(1) * n/B = O(2n/B) = O(n/B)$

4. Derive the general formula for work and span, assuming a k^2-way tableau construction (i.e., the tableau is divided up into k^2 pieces of size n/k × n/k).

```
#define A(i, j) A[N + (i) - (j) - 1]

 void recursive_tableau(double *A, size_t rbegin, size_t rend, size_t cbegin,
 size_t cend) {
     if (rend-rbegin == 1 && cend-cbegin == 1) {
         size_t i = rbegin, j = cbegin;
         A(i, j) = f(A(i-1, j-1), A(i, j-1), A(i-1, j));
     } else {
         size_t rmid = rend-rbegin > 1 ? (rbegin + (rend-rbegin) / 2) : rend;
         size_t cmid = cend-cbegin > 1 ? (cbegin + (cend-cbegin) / 2) : cend;
         recursive_tableau(A, rbegin, rmid, cbegin, cmid);
         if (cend > cmid)
             recursive_tableau(A, rbegin, rmid, cmid, cend);
         if (rend > rmid)
             recursive_tableau(A, rmid, rend, cbegin, cmid);
         if (rend > rmid && cend > cmid)
             recursive_tableau(A, rmid, rend, cmid, cend);
     }
 }

for(size_ti=0; i<N; i++){
    A(i, 0) = INIT_VAL;
}

for(size_tj=0; j<N; j++){
    A(0, j) = INIT_VAL;
}

if (N >1) {
    recursive_tableau(A, 1, N, 1, N);
}

res = A(N-1, N-1);
```

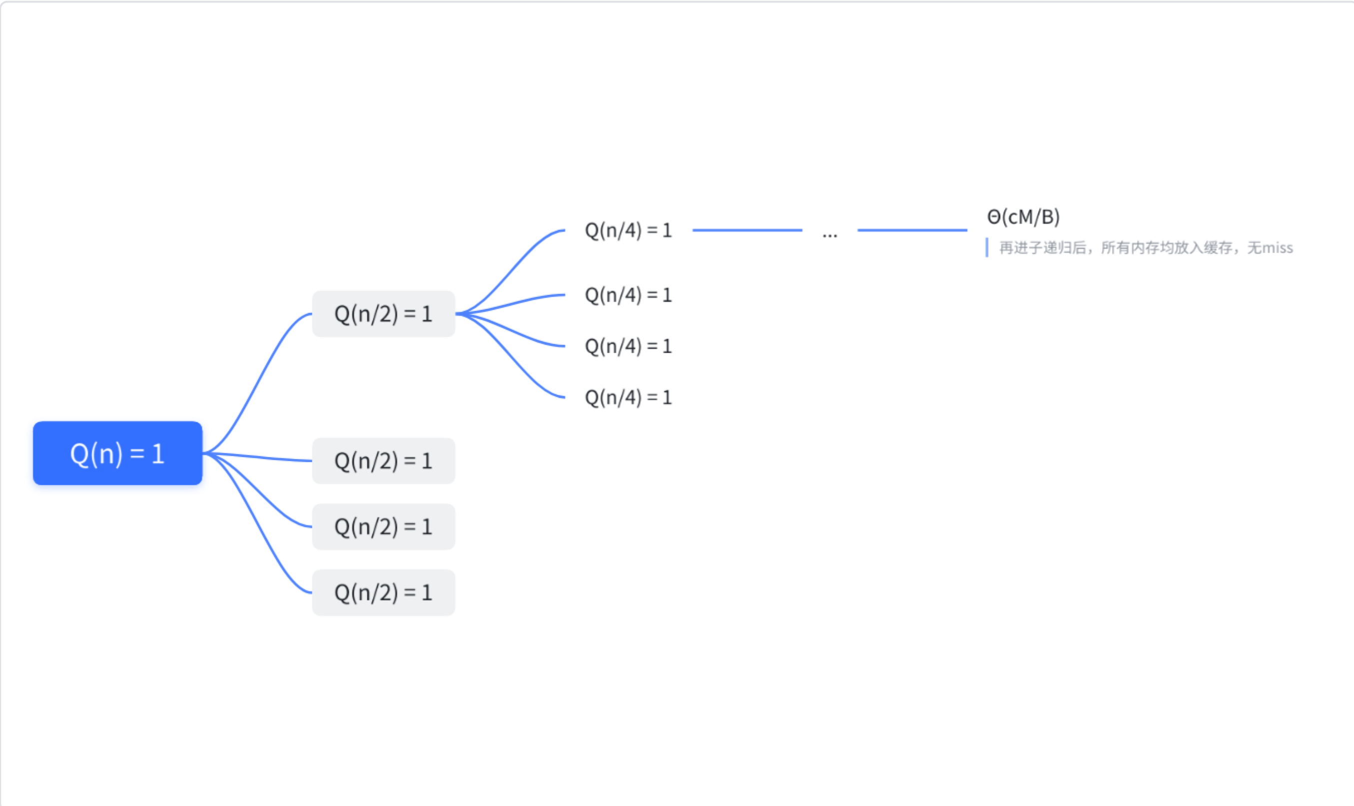
> work:
> $W(n) = k^2 * W(n/k) + \Theta(1)$.根据主定理有$n^{log_k(k^2)} = n^2, f(n) = \Theta(1)$.故$W(n) = \Theta(n^2)$
> span: $S(n) = k^2 * S(n/k) + \Theta(1) \; = \; \Theta(n^2)$

5. Answer the following questions assuming that an optimal replacement strategy holds and that the cache is tall.

- Show the recurrence relation for the cache complexity Q(n) using the 4-way construction of the recursive_tableau function.

> $$Q(n) = \begin{cases} \Theta(cM/B) & \text{if } 2n-1 < cM \\ 4Q(n/2) + O(1) & \text{otherwise} \end{cases}$$

- Draw the recursion tree and label the internal nodes and leaves with their cache complexity Q(n). What’ s the height of the recursion tree?

> 
>
>
> *当 cM = 2x - 1时cache miss递归树不再触发miss，x = (cM+1)/2。*
>
> *故树的高度为* $log_2 n - log_2(cM+1)/2$

- How many leaves are in the recursion tree?

> *叶子树数量：* $4^{log_2 n - log_2(cM+1)/2} = \Theta(n^2/M^2)$
>
> *换底公式推演：* $4^{log_2 n} = (2^2)^{log_2 n} = 2^{2 log_2 n} = 2^{log_2(n^2)} = (n^2)^{log_2 2} = n^2$

- Using the recursion tree and the recurrence relation, derive a simplified expression for Q(n).

> $\Theta(n^2/M^2) * cM/B = \Theta(n^2/MB)$

6. Answer the following question assuming that an optimal replacement strategy holds and that the cache is tall. Assuming a k^2-way tableau construction, show that if we are "unlucky," where a subpiece is just slightly above the cache size, then we have Q(n) = Θ(n^2k/MB). Also show that if we are lucky and this situation does not arise, then we have Q(n) = Θ(n^2/MB).

> - *lucky*
>
> *树的高度为* $log_k n - log_k((cM+1)/2)$
>
> *叶子树数量：* $(k^2)^{log_k n - log_k(cM+1)/2)} = \Theta(n^2/M^2)$
>
> *故总cache miss* $Q(n) = \Theta(n^2/M^2) * cM/B = \Theta(n^2/MB)$
>
> - *unlucky*
>
> 根据假设，当前理想叶子块应该是大小为cM，但是实际的的树level（整数）的块大小可能略微比cM大一点，无法缓存整个块，此时需要树再扩展一层。
>
> 扩展一层后的块大小近似为cM/k，且一定小于cM，满足将整个块放入缓存中。此时叶子节点是上一层节点的k^2倍。根据lucky公式
>
> 此时叶子树数量：$\Theta(n^2/M^2) * k^2 = \Theta(n^2k^2/M^2)$
>
> 此时子块的$cache\ miss$为： $cM/k/B = cM/Bk$
>
> 此时总体$cache\ miss$为： $\Theta(n^2k^2/M^2) * cM/Bk = \Theta(n^2k/MB)$

空白 TeX 公式